Storing Shortest Paths for a Polyhedron

Jindong Chen and *Yijie Han*

Department of Computer Science
University of Kentucky
Lexington, KY 40506

Abstract

We present a new scheme for storing shortest path information for a polyhedron. This scheme is obtained with a new observation on the properties of shortest path information of a polyhedron. Our scheme separates in a clear sense the problem of finding shortest paths and the problem of storing the shortest path information for retrieval. A tradeoff between time complexity $O(d \log n / \log d)$ and space complexity $O(n \log n / \log d)$ is obtained, where d is an adjustable parameter. When d tends to infinity space complexity of $o(n \log n)$ can be achieved at the expense of increased time complexity.

1 Introduction

The single source shortest path problem on the surface of a polyhedron has been studied by a number of researchers[CH][MMP][M1][M2][SS]. Recent result[CH] reveals that the single source shortest path problem can be solved in $O(n^2)$ time and $\Theta(n)$ space. Sharir and Schorr[SS] showed that by solving the single source shortest path problem for a convex polyhedron, the solution can be stored such that, for a query point $p = (f, c)$, where f is the face point p lies on and c is the position of p on f, the shortest distance from the source to p may be computed in $O(\log n)$ time and the edge sequences of the shortest path can be

computed in $O(\log n + k)$ time, where k is the number of edges in the edge sequence. The information stored by Sharir and Schorr's algorithm is the subdivisions of the faces of the polyhedron. The subdivisions are, to a large degree, the immediate result of computing the shortest paths in their algorithm. Since their algorithm uses $O(n^2)$ space to compute $O(n^2)$ subdivisions, the solution is consequently stored using $O(n^2)$ space. Mount showed that the problem of storing shortest path information can be treated separately from the problem of computing the shortest paths. He showed that the shortest path information for a convex polyhedron can be stored in $O(n \log n)$ space while maintaining $O(\log n)$ time for a query of shortest distance and $O(\log n + k)$ time for a query of the edge sequence of the shortest path. What Mount observed is that the edges of the polyhedron, after certain processing, form a total order according to their "distance" from the source (we shall refer to this as the Mount order). The Mount order, in turn, can be used to mimic the shortest path edge sequences starting from the source point and extending outwards[M1]. Mount used these sequences to build a set of trees for storing the shortest path information.

For a nonconvex polyhedron no result on the space complexity better than $O(n^2)$ has been claimed, although Mount's scheme[M1] may be extendible to nonconvex polyhedrons.

What remains unsolved is the question of whether the space complexity can be reduced further? In view of many known storing methods for various computational geometry problems, Mount[M1] and Mitchell, Mount and Papadimitriou[MMP] raised the open problem of whether space complexity of $O(n)$ can be achieved for storing shortest path information of a polyhedron?

In this paper we observe that the essential part of the shortest path information is the transformations represented by the notches in the planar layout of the polyhedron. Our observation leads to a fairly simple and clear interpretation of storing the transformations of the notches of the polyhedron (which need not be convex). When combined with the Mount order[M1] on the layout of a polyhedron, it provides a rather uniform scheme of storing the shortest path information in one tree which demonstrates the tradeoff between the space complexity (for storing the information) and the time complexity (for answering a query). Our scheme uses $O(n \log n / \log d)$ space for storing the shortest path information, $O(d \log n / \log d)$ time for processing a query of the shortest distance from the source point to the query point, $O(d \log n / \log d + k)$ time for computing the edge sequence of the shortest path, where k is the number of edges in the edge sequence of the shortest path and d is an adjustable parameter (which is the degree of the tree storing the shortest path information). For example, when d is a constant, our scheme yields the same result obtained by Mount[M1].

When $d = f(n)$ tends to infinity, the space used is $o(n \log n)$. When $d = n^{\epsilon}$, $0 < \epsilon \leq 1$, the space is reduced to $O(n)$ while time is increased to $O(n^{\epsilon})$.

For a simple exposition, the discussion of our scheme is aimed at a convex polyhedron. Because our scheme is based on the technique of coalescing and the Mount order, and because both can be trivially extended to the nonconvex case, the time complexity $O(d \log n / \log d)$ and the space complexity $O(n \log n / \log d)$ can also be achieved by our scheme for a nonconvex polyhedron.

The rest of the paper is organized as follows. In section two we explain the two layouts of a convex polyhedron, one is due to Sharir and Schorr[SS], the other is due to Chen and Han[Ch][1]. We also explain the Mount order of the edges of a polyhedron. This order is due to Mount[M1]. We give a simple scheme for storing shortest path information for a convex polyhedron in $O(n^2)$ space by using the two layouts. This scheme, although not efficient, provides an alternative approach to the problem. In section three we present our main result, namely the scheme of storing the shortest path information with stated space and time complexities. Conclusions are drawn in section four.

2 Layouts and the Mount Order

The planar layout of the surface of a convex polyhedron due to Sharir and Schorr is obtained by cutting all the ridge lines[SS]. Ridge lines are the loci of ridge points while a ridge point is a point for which there are more than one shortest paths from the source point. After cutting the ridge lines the surface of the polyhedron can be laid out on a plane. This layout is a star shaped polygon with the edges consisting of ridge lines. The vertices of the layout polygon are either vertices of the polyhedron or the Voronoi vertices on the ridge lines(after being transformed into the layout plane). If two ridge lines intersect each other at a vertex, they form a notch. An example of such a layout is illustrated in Fig. 1.

There is another layout of the surface of a polyhedron due to Chen and Han[CH]. This layout is obtained by cutting the shortest paths from the source point to the vertices of the polyhedron. This layout is also a star shaped polygon with edges consisting of the shortest paths from the source point to vertices of the polyhedron. The vertices of the polygon are either the vertices of the polyhedron (after being transformed into the plane) or the images of the source point (there are $O(n)$ points which are images of the source point). Note that

[1] Agarwal *et al.* discovered the layout independently.

when a vertex of the polyhedron is also a ridge point, the shortest paths to this vertex form closed curves. A notch is formed in the layout when a shortest path to a vertex is cut. An example of such a layout is shown in Fig. 2.

We shall call the layout due to Sharir and Schorr outward layout (for shortest paths emanate from the center, *i.e.* the source point, outwards toward the destinations) and the layout due to Chen and Han inward layout (where a shortest path from the source to a destination going inward the interior of the layout polygon).

In a layout the vertices of the polyhedron except the images of the source point (which can be a vertex of the polyhedron as well) can be arranged in a circular order. In the outward layout this circular order has been shown by Sharir and Schorr[SS] and Mount[M1]. The same circular order used in the outward layout can be used in the inward layout to order the vertices of the polyhedron. This is obvious because if we cut along the shortest paths to the vertices of the layout polygon in the outward layout (all these shortest paths are contained within the layout polygon) and then coalesce along the ridge lines of the layout polygon we obtain the inward layout. Observe that if we assume that the circular order in the outward layout is counter-clockwise then the circular order of the inward layout is clockwise.

Suppose we connect the adjacent vertices in the circular order by a straight line segment in the layouts (in case of a nonconvex polyhedron, they could be hyperbolas). These line segments form a closed curve. This closed curve decomposes each layout polygon into two parts, one contains the source (we shall call it the arctic) while the other does not contain the source (we shall call it the antarctic). We shall call this closed curve the equator of the polyhedron with respect to the source point. Note that the equator is completely determined by the polyhedron and the source point. This equator can be used to devise an alternative scheme for storing the shortest path information for a convex polyhedron.

Consider the faces of the polyhedron. Each face could be cut by the equator into several regions. Each region is either in the arctic or in the antarctic. We associate each region with a transformation T which transforms the region to the right position on a layout. If the region is in the arctic, we associate it with a T which transforms it to the place on the outward layout. If the region is in the antarctic, we associate it with a T which transforms it to the place on the inward layout. If the information of the cutting of faces by the equator and the transformations associated with each cut region by the equator is stored, we can easily determine the shortest path for each query point on the surface of the polyhedron as follows. For a query point p on face f, first determine the region R point p lies on. If R is in the arctic we use the transformation T associated with R to transform R to the plane

of outward layout, otherwise use T to transform R to the plane of inward layout. From the layouts we can determine the shortest distance and the orientation of the shortest path. This scheme requires $O(n^2)$ space to store the needed information. Note that we do not store the cutting of the faces by the ridge lines. We store the cutting of the faces by the equator. This scheme does not use less space than known schemes, it merely provides an alternative view to the problem of storing the shortest path information.

We now give a brief review of the ordering of the edges of the polyhedron obtained by Mount[M1]. The first step in obtaining the ordering is to decompose each edge e of the polyhedron into two directed edges $\overleftarrow{e}$ and $\overrightarrow{e}$. For a shortest path $\overrightarrow{p}$ (which is a ray emanating from the source) crossing e, we say that $\overrightarrow{p}$ crosses edge $\overrightarrow{e}$ if the angle formed from $\overrightarrow{p}$ to $\overrightarrow{e}$ is less than π, otherwise we say that $\overrightarrow{p}$ crosses $\overleftarrow{e}$. With this convention the shortest paths on the outward layout always cross the edges from the left to the right. To obtain the total order of the (directed) edges Mount also split certain directed edges in order to break the possible spirals formed in ordering the edges. For a detailed description the readers are referred to Mount[M1].

For our purpose we use the result that the edges can be ordered on the outward layout, as to how to order the edges is not intimately related to our storing scheme. In the rest of the paper we simply assume that the (directed) edges (those that are present in the outward layout) form a total order in the outward layout.

3 A Scheme for Storing Shortest Path Information

We shall use the outward layout and the Mount order to store shortest path information. Recall that the outward layout is a polygon. The vertices of the layout polygon are vertices of the polyhedron and the vertices of the Voronoi diagram. There are a total of $O(n)$ vertices. We connect each of these vertices with the source by the shortest paths to the source. We thus obtain $O(n)$ ordered "triangles" by the circular order as shown in Fig. 3.

Note that the inward layout can be obtained by first cutting off these triangles from the outward layout and then coalescing them. This coalescing process can be viewed as transformations which associate each triangle with a transformation matrix. By the circular order of the triangles, the coalescing process can proceed in the following manner. Take the first triangle, coalesce it with the second triangle, then coalesce them with the third triangle, then coalesce with the fourth triangle, and so on.

Now consider two adjacent triangles T_1 and T_2. They can be related by a transformation matrix which coalesce T_1 with T_2. Take an edge $\vec{e}$ of the polyhedron and observe how $\vec{e}$ is laid out on the outward layout. $\vec{e}$ could traverse between the arctic and the antarctic (which is separated by the equator). For a subsegment e of $\vec{e}$ which is laid out in the arctic, e is continuous (we use the word *continuous* to mean that the edge is not broken by notches, a *noncontinuous* edge or a *broken* edge is an edge going through some notches). For a subsegment e' laid out in the antarctic, e' is broken up by the edges of the layout polygon. However, we can coalesce the triangles to obtain a continuous e'. (In fact certain parts of e' could still be missing because they are not present in the outward layout, this is a consequence of decomposing the edges into directed edges and forming the total order[M1]. These missing parts can be viewed, at least conceptually, as present. This phenomenon of missing parts does not affect the correctness of our scheme.)

We have essentially outlined a technique by which we can use transformation matrices to coalesce certain triangles to obtain a continuous edge of the polyhedron. Suppose M_1, M_2, $\cdots$, M_k are the transformation matrices used to obtain the continuous edge $\vec{e}$. For a point p on $\vec{e}$ we can use binary testing to find out on which triangle p lies. This is done by first transforming the tail of $\vec{e}$ from the polyhedron onto the outward layout, then by multiplying the transformed point p by transformation matrices M_1 through $M_{\lfloor k/2 \rfloor}$ to see whether p lies to the left, right or on the last triangle T coalesced. If p is on T then we are done because the triangle gives the distance of the shortest path and also the orientation of the shortest path which can be used to find out the edge sequence passed by the shortest path (it is actually done by transforming the orientation back to the polyhedron and then trace out the shortest path on the surface of the polyhedron). If p is to the left of T then we have overdone it, we should have multiplied fewer matrices. We undo what we have done and in the next step we will multiply p by transformation matrices M_1 through $M_{\lfloor k/4 \rfloor}$. If p is to the right of T then in the next step we will continue to multiply the transformed p by matrices $M_{\lfloor k/2 \rfloor+1}$ through $M_{\lfloor 3k/4 \rfloor}$. This process continues until we find the triangle p lies on. The matrix-to-matrix multiplications need not be done at the time of processing queries, they can be precomputed and stored, only to be used in the processing of queries.

The technique presented here would allow us to process the shortest path queries for points on a single edge of the polyhedron in $O(\log n)$ time using $O(n)$ storage. For different edges different sets of transformation matrices are needed to coalesce triangles. This is because different edges may have different patterns of entering and exiting the arctic and the antarctic. If we store different transformation matrices for each edge of the polyhedron using the above method we will end up with space complexity of $O(n^2)$. To reduce the space

complexity we use Mount order.

Suppose edge $\vec{e_1} > \vec{e_2}$, *i.e.*, $\vec{e_1}$ is "farther away" from the source than $\vec{e_2}$ in the Mount order. If transformation matrix M is not used to coalesce triangles for $\vec{e_1}$, then it is not used for $\vec{e_2}$. This situation is illustrated in Fig. 4. In general, if there are k consecutive transformation matrices and $n \leq 2k$ edges, then the number of patterns of coalescing triangles for these edges is $2k$.

Example: 2 consecutive transformation matrices M_1, M_2 form 4 patterns of coalescing. The possible situations are as follows.

1). I, M_1, M_1M_2, M_{1-2}.

2). I, M_2, M_1M_2, M_{1-2}.

I is the identity matrix. The value of M_{1-2} is equal to that of M_1M_2 although they represent different patterns. This example is illustrated in Fig. 4. □

Note that we have to distinguish the pattern of M_{1-2} from the pattern of M_1M_2 because in the pattern represented by M_{1-2} we can not coalesce the first triangle with the second triangle by multiplying M_1 and then check whether the query point is to the left, right, or on the second triangle. For a query point on the edge represented by the pattern M_{1-2}, we have to coalesce the first triangle with the second triangle and the third triangle at the same time. In this case we say that the edge jumps through the second triangle.

We store transformation matrices into a tree. The transformation matrices associated with notches of the layout polygon are stored at the leaves of the tree from left to right by their circular order. Each leaf of the tree is associated with a notch of the outward layout polygon. A d-ary tree is used. Let a be an interior node of the tree. If the subtree rooted at a has k leaves we store $2k-1$ transformation matrices at node a, each of them representing a pattern of coalescing the triangles which form the notches at the leaves of the subtree rooted at a. Note that we do not store the pattern of identity transformation. In addition, we store the information of edges jumping through the triangle between the notch at the rightmost leave of the subtree rooted at a and the notch at the leftmost leave of the subtree rooted at the immediate right sibling of a. Since there are $O(n)$ leaves in the tree which represent $O(n)$ notches in the outward layout, the tree has height $O(\log n / \log d)$. For each node a at level 0 of the tree (a is a leaf) we store one transformation matrix associated with the notch represented by a. For an interior node a at level i, we store $2d^i - 1$ matrices at a because the subtree rooted at a has d^i leaves and therefore it has at most $2d^i - 1$ patterns of coalescing represented by nonidentity transformation matrices. Since there are $O(n/d^i)$

nodes at level i, the total number of matrices stored at level i is $O(n)$. Therefore the total number of matrices stored in the tree is $O(n \log n / \log d)$.

For a query point on an edge of the polyhedron, we first use a transformation matrix associated with the tail of the edge to transform the tail into the plane of outward layout. We then use the tree to locate the triangle the query point lies on. At an interior node a, we test the transformations associated with the children of a one by one from the leftmost child of a to the rightmost child of a to find out which child of a should we continue the search. Since a has d children, it takes $O(d)$ tests to locate the child for the continuing search. Because the tree has $O(\log n / \log d)$ levels, the total number of tests for locating the triangle where p lies on is $O(d \log n / \log d)$. It remains to show that these tests can be done in $O(d \log n / \log d)$ time.

We describe in detail the data structure used to store the transformation matrices. We denote by $M_i^{j,k}$ the product of the matrices used to transform the i-th edge from the j-th triangle to the m-th triangle such that $m \leq k+1$ and m-th triangle is the largest indexed triangle edge i passes through (instead of jumping through). Call $[j,\ k+1]$ the range of the transformation. We also say that j-th notch(leaf) to the k-th notch(leaf) are covered by the transformation. For each $M_i^{j,k}$ we use an auxiliary matrix $N_i^{j,k}$ which is the product of the matrices from the m-th notch all the way to the k-th notch. When $m = k+1$ $N_i^{j,k} = I$. It is obvious that $M_i^{j,l} = M_i^{j,k} N_i^{j,k} M_i^{k+1,l}$ for $j < k < l$. We call $M_i^{j,k}$ a boundary matrix if $M_i^{j,k} \neq I$ and $M_i^{j,k}$, $M_{i-1}^{j,k}$ represent different patterns of coalescing. Matrices stored at node a are those boundary matrices and their auxiliary matrices each has the range of transformation covering the leaves of the subtree rooted at a.. Boundary matrices and their auxiliary matrices stored at node a are ordered by their subscripts and placed into an array. $M_{i1}^{j,k}$ is stored before $M_{i2}^{j,k}$ if $i1 < i2$, $N_i^{j,k}$ is stored next to $M_i^{j,k}$. To facilitate searching on the tree we add two pointers to each boundary matrix stored in the tree. Let $M_i^{j,k}$ be a boundary matrix stored at node a. Let $M_{p_1}^{j,k_1}$, $M_{p_2}^{j,k_1}$, ..., $M_{p_q}^{j,k_1}$ be the boundary matrices stored at the leftmost child b of a, in that order. Let $M_{u_1}^{k+1,k_2}$, $M_{u_2}^{k+1,k_2}$, ..., $M_{u_v}^{k+1,k_2}$ be the boundary matrices stored at the immediate right sibling c of a, in that order. We use one pointer (the child pointer) for $M_i^{j,k}$ to point to $M_{p_t}^{j,k_1}$, where $M_{p_t}^{j,k_1}$ satisfies that $p_t \leq i < p_{t+1}$. In case $p_1 > i$ the pointer will point to $M_{p_1}^{j,k_1}$. The information provided by this pointer is the position among the matrices stored at b where M_i^{j,k_1} is to be inserted should we insert M_i^{j,k_1} into the matrices at b. Another pointer (the sibling pointer) points to the boundary matrix $M_{u_w}^{k+1,k_2}$ which satisfies $u_w \leq i < u_{w+1}$. In case $u_1 > i$ we let this pointer point to $M_{u_1}^{k+1,k_2}$. The information provided by this pointer is the position among the matrices stored at c where M_i^{k+1,k_2} would be inserted. We have yet to make an assumption in order

to achieve $O(d \log n / \log d)$ time. We assume, for any two consecutive matrices $M_{i1}^{j,k}$ and $M_{i2}^{j,k}$ stored at a, that the number of matrices stored at c with indices between $i1$ and $i2$ (inclusive) (call it the number of matrices covered by $M_{i1}^{j,k}$ and $M_{i2}^{j,k}$) is no more than three. With this assumption we can trace the pointers in the tree to find the triangle the query point p lies on in time $O(d \log n / \log d)$. Assuming that the query point p is on edge $i3$ with $i1 < i3 < i2$ (the Mount order) and, the current node is a, the current matrix is $M_{i1}^{j,k}$, what is needed in moving from a to c is to find the index u_w satisfying $u_w \leq i3 < u_{w+1}$, among the indices of matrices at c which are between $i1$ and $i2$. Since there are no more than three such matrices the index u_w can be found in constant time.

To remove the assumption in the last paragraph we use the technique of padding. For each level of the tree we start from the rightmost node to the leftmost node. Assume that our assumption already holds for nodes at the same level and to the right of a. For two consecutive matrices $M_{i1}^{j,k}$ and $M_{i2}^{j,k}$ at a, let u_1, u_2, ..., u_w be indices of matrices at c which are between $i1$ and $i2$, where c is the immediate right sibling of a. If $w > 3$ we pick every second indices from the sequence u_1, u_2, ..., u_w (from the sequence u_2, u_3, ..., u_w if $i1 = u_1$) and insert matrices $M_{u_2}^{j,k}$, $M_{u_4}^{j,k}$, ..., ($M_{u_3}^{j,k}$, $M_{u_5}^{j,k}$, ..., if $i1 = p_1$) between $M_{i1}^{j,k}$ and $M_{i2}^{j,k}$ at node a. The same process is performed for all consecutive matrices at a. After this process any two consecutive matrices at a covers at most three matrices at c. Since each padding matrix is added for at least three existing matrices at c, the number of padding matrices added into the whole tree is no more than a constant multiple of the number of original matrices in the tree which is $O(n \log n / \log d)$.

Example: The two dimensional array shown in Fig. 5 gives the relation between edges and notches of an assumed outward layout. The x dimension is the notches and the y dimension is the edges. If the entry (i, j) is $-$, then the edge j is broken by notch i. Otherwise it is not broken by notch i. A $-$ between two $-$'s indicates that the triangle between the two notches is jumped through by the edge. The corresponding binary tree for storing shortest path information is also shown in Fig. 5. Note that auxiliary matrices are not shown in the figure although they are stored in the tree. □

The scheme we explained above allows us to process queries with destination point p lying on edges. To process queries with arbitrary destination point on the surface of the polyhedron we have to first solve the end region problem.

We may assume that the faces of the polyhedron are triangles[CH][MMP][M1][M2]. For a triangle t on the outward layout we say t ends within face f if t intersects only one of the three edges of f. If t ends within f $t \cap f - \{\partial t \cup \partial f\}$ is called the end region of t, where ∂t

and ∂f are the edges of t and f, respectively. If t intersects all three edges of f we say that t occupies f. If t has x end regions t must occupy $x-1$ facess. Because a face f can be occupies by at most one triangle, the total number of end regions for a polyhedron is $O(n)$.

For a query point p not on an edge the shortest path to p must pass one of the three edges of the face f p lies on. If we test the query point using the coalescing patterns of these edges, we will have problem if there are triangles end within face f, for if we have coalesced to such a triangle we would have no way of knowing whether the query point is to the left or right of the triangle. We solve this problem by the following scheme. We cut end regions out from both the faces and the triangles and store them separately. Now each triangle t ends at an edge e. In constructing the search tree we treat edge e as if it jumps through triangle t.

Now for a query point p on face f we first test whether it is within an end region on face f. This can be done in $O(\log n)$ time using known point location techniques[LT]. If it is not in an end region we then test the query point by trying the coalescing pattern of each of the edges of f which are present in the outward layout (if the directed edge is not present, no shortest path crossing this edge exists), the edge which yields the shortest distance for point p is the edge the shortest path crosses.

We have proved the following theorem.

Theorem: The information for the single source shortest paths on the surface of a polyhedron can be stored in $O(n \log n / \log d)$ space which supports the processing of a query of shortest distance in $O(d \log n / \log d)$ time, and the processing of a query of shortest path in $O(d \log n / \log d + k)$ time, where k is the number of edges the shortest path passes through and $1 < d \leq n$ is an adjustable integer. □

4 Conclusions

We have presented new schemes for storing shortest path information for a polyhedron. One scheme uses the equator and the other scheme uses the method of coalescing triangles. Both schemes provide new insight to the problem of storing the shortest path information. The question remains open whether simultaneous achievement of space complexity of $O(n)$ and time complexity of $O(\log n)$ is possible.

We note that the inward layout seems to be simpler than the outward layout for there is no edge jumping through triangles. We use the outward layout to store information because

Mount order helps to cut the space complexity. It is of interest to us whether a total order on the inward layout similar to Mount order can be found to help the construction of a search tree from the inward layout.

References

[AAOS] P. Agarwal, B. Aronov, J. O'Rourke, and C. Schevon. Star unfolding of a polytope with applications. Proc. of the Scandanavian Workshop on Algorithm Theory. LNCS 447, 1990.

[CH]. J. Chen and Y. Han. Shortest paths on a polyhedron. Proc. of 1990 ACM Symposium on Computational Geometry, 360-369.

[LT]. Lipton, R. and R. Tarjan. Applications of a planar separation theorem, SIAM J. Comput. 9(3) (1980), 478-487.

[MMP]. J. S. B. Mitchell, D. M. Mount, C. H. Papadimitriou. The discrete geodesic problem. SIAM J. Comput., Vol. 16, No. 4, 647-668(Aug. 1987).

[M1]. D. M. Mount. On finding shortest paths on convex polyhedra. Tech. Report 1496, Dept. of Computer Sci., Univ. of Maryland, Baltimore, MD, 1985.

[M2]. D. M. Mount. Storing the subdivision of a polyhedral surface. Proc. 2rd ACM Symposium on Computational Geometry, Yorktown Heights, NY, June 2-4, 1986.

[SS]. M. Sharir and A. Schorr. On shortest paths in polyhedral spaces. SIAM J. Comput., 15(1986), pp. 193-215.

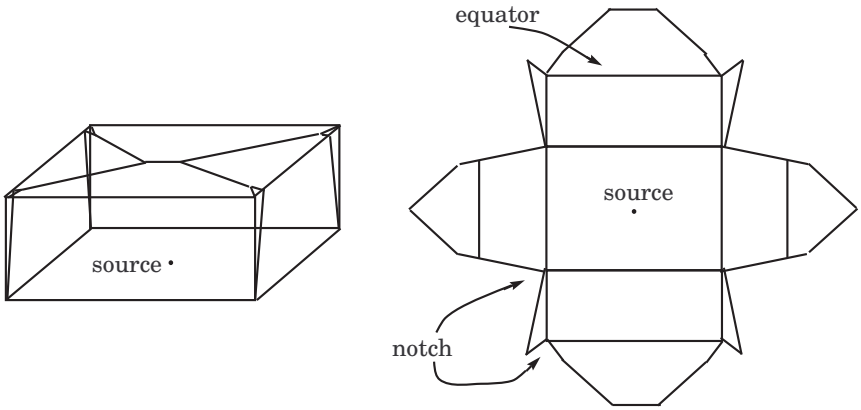

Fig. 1. A polygon and its outward layout

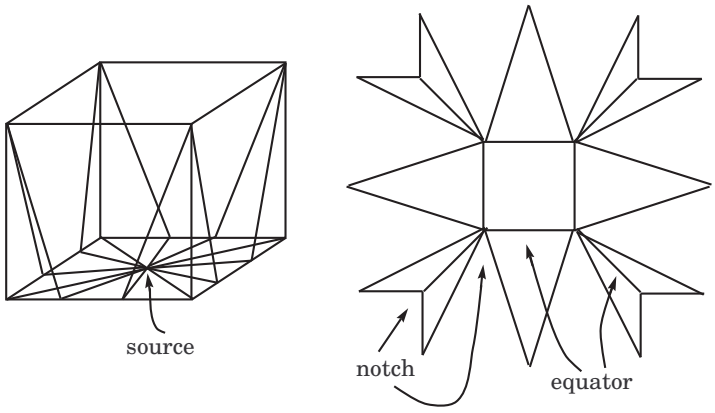

Fig. 2. A polygon and its inward layout

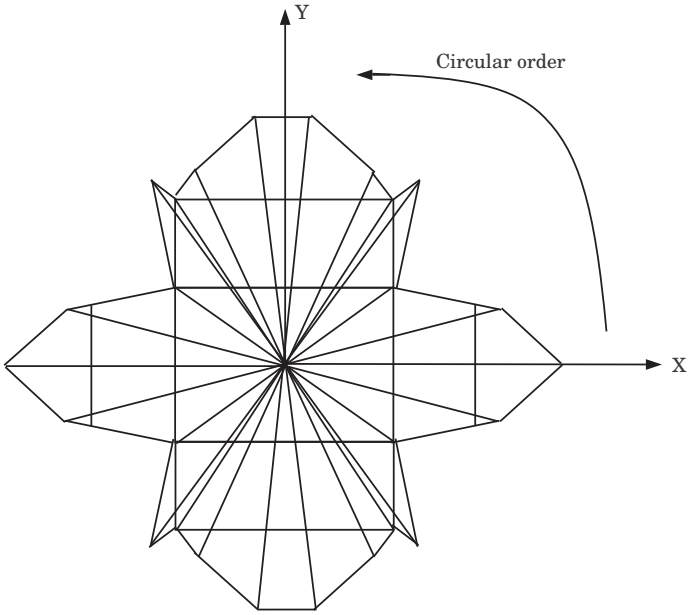

Fig. 3. Triangulation and the circular order

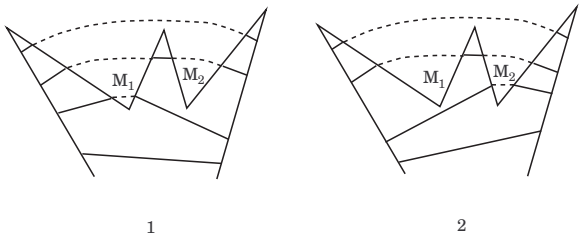

Fig. 4. Two situations of coalescing each has four patterns

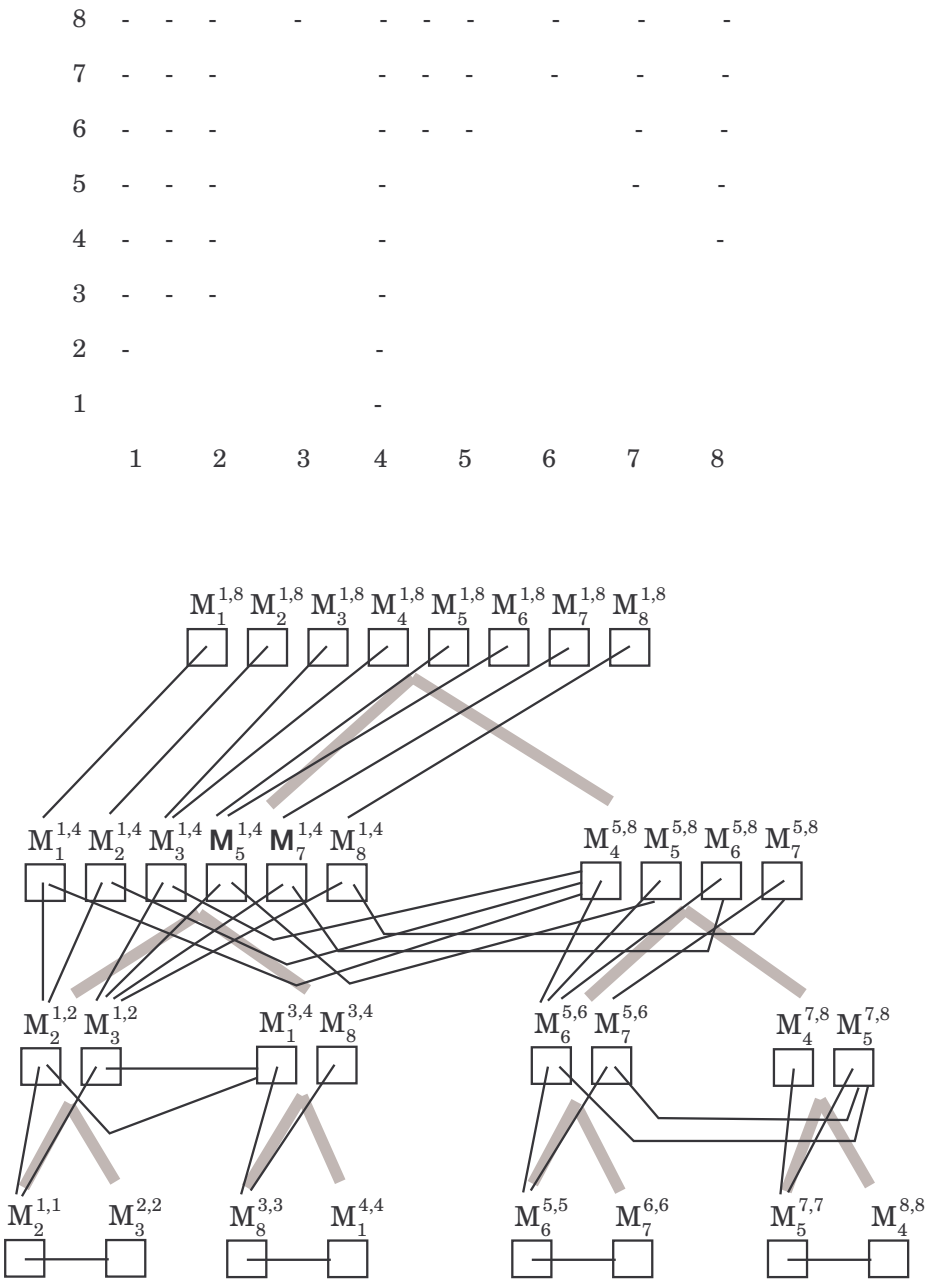

Fig. 5. Storing the coalescing patterns in the search tree